1) (0.5 puntos) Conceptos generales:

a. Escribir una definición de programación concurrente y programación paralela, diferenciando ambos conceptos.

Un programa concurrente especifica 2 o + programas secuenciales que pueden ejecutarse concurrentemente en el tiempo. Puede tener N procesos habilitados para ejecutarse y disponer de M procesadores, c/u de los cuales puede ejecutar 1 o + procesos.

El procesamiento paralelo es la ejecución concurrente en múltiples procesadores que cooperan entre sí, con el objetivo de mejorar el rendimiento (tiempo de ejecución) respecto al algoritmo secuencial. Necesita que cada hilo / proceso se ejecute sobre su propia unidad de procesamiento.

b. Definir lo que es la sincronización entre procesos y explicar cuáles son los dos tipos de sincronización en que se clasifica. Dar un ej (explicarlo con palabras, no con código) de un problema donde se usen ambos tipos de sincronización.

Sincronización entre procesos: posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan por:

- Exclusión mutua: asegura que solo 1 proceso tenga acceso a un recurso compartido en un instante de tiempo. Evita que 2 o + procesos puedan encontrarse en la misma SC al mismo tiempo. Ej: si se tiene un recurso que puede usarse por 1 sólo proceso a la vez, se utilizan exclusión mutua para asegurarnos de que se cumpla.
- Condición: permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada. Ej: antes de comenzar, 1 proceso chequea que todos los demás procesos hayan llegado al mismo punto de ejecución.

EJ de ambos: un sistema de productor / consumidor:

- ✓ Exclusión mutua: sólo 1 proceso puede acceder al buffer compartido a la vez para agregar o retirar elementos, evitando conflictos.
- ✓ Condición: el productor debe esperar si el buffer está lleno, y el consumidor debe esperar si el buffer está vacío, hasta que el otro proceso realice la acción necesaria (producir o consumir un elemento).

c. Definir comunicación entre procesos. Explicar cuáles son los dos tipos de comunicación que existen.

Comunicación entre procesos: el tipo de comunicación indica cómo se organizan y transmiten los datos. Los procesos se comunican por:

- Memoria compartida: los procesos intercambian info sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. No pueden operar simultáneamente sobre la memoria, obligando a bloquear y liberar el acceso a la misma.
- Pasaje de mensajes: se establece un canal para transmitir información entre procesos. Los procesos deberán saber cuándo tienen mensajes para leer o para transmitir.

d. Indicar qué es la programación distribuida y cuáles son las características que lo diferencian de la programación con memoria compartida.

Programación distribuida: aquella en la que los procesos ejecutan un programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).

A diferencia de la programación con memoria compartida, en la distribuida se utilizan mensajes, en vez de variables compartidas, para la sincronización y comunicación, ya que SOLO comparten canales. Debido a la transmisión de datos a procesos alocados externamente, es + lenta que la programación con memoria compartida.

2) (1.5 puntos) Problema de la sección Crítica:

a. Nombrar y explicar las 4 propiedades que debe cumplir una solución a este problema. Para cada una de estas 4 propiedades explicar la idea de una solución que no cumpla dicha propiedad (con palabras, no con código).

Propiedades que deben satisfacer los protocolos de E/S a una SC (PS -> prop seg - PV -> vida):

1. Exclusión Mutua (PS): como máximo 1 proceso está en su SC en un momento dado.
2. Ausencia de Deadlock (Livelock) (PV): si 2 o + procesos intentan entrar en sus SC, al menos 1 tendrá éxito. Básicamente evitar que los procesos queden bloqueados en un punto muerto y no puedan avanzar.
3. Ausencia de Demora Innecesaria (PV): evitar espera innecesaria de un recurso ya disponible.
4. Eventual Entrada (PV): garantizar que un proceso eventualmente entrará en su SC, evitando la inanición.

b. Desarrollar una solución de GRANO FINO usando sólo variables compartidas (no usar await, sentencias especiales como TS, semáforos o monitores) usando un coordinador. En este caso, cuando un proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le dé permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador.

```
1  llegue[N]=([N] false)
2  puedo[N]=([N] false)
3  int i=0
4  process Coordinador{
5      while(true){
6          while(not llegue[i]) i=(i+1) mod N
7          puedo[i]=true
8          while(llegue[i]) {skip}
9      }
10 }
11 process Proceso[id=0..N-1]{
12     --SNC
13     llegue[id]=true
```

3) (1 punto) Monitores: indicar como se realiza en monitores la comunicación y la sincronización (ambos tipos de sincronización) entre procesos. Explicar la diferencia entre los protocolos de sincronización en monitores: signal and wait (S&W) y signal and continue (S&C).

Comunicación entre procesos: será haciendo uso del monitor que va a contener los datos compartidos, que son variables o estructuras de datos accesibles por varios procesos. Dentro del monitor van a haber variables condicionales que permitirán a los procesos esperar o notificar eventos específicos.

Sincronización:

- Exclusión mutua: está dada, es implícita. Un monitor va a estar siendo utilizado en 1 momento solo por 1 proceso. Los demás se encolan y cuando el proceso termine de ser usado, se va a elegir 1 encolado de forma no determinística.
- Por condición: es con variables condición -> cond cv;. cv es una cola de procesos demorados, no visible directamente al programador. Operaciones: wait (cv), signal (cv), signal_all (cv). Operaciones adicionales: empty (cv), wait (cv, rank), minrank (cv).

S&W y S&C: la diferencia principal entre ambas disciplinas recae en quién es el que va a ser encolado nuevamente para esperar el uso del monitor. En el caso de Signal and continue va a ser encolado (en la cola no determinística) el proceso despertado y en Signal and wait va a ser encolado el proceso que despierta.

Signal and continue: el proceso que hace el signal continúa usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al wait).

Signal and wait: el proceso que hace el signal pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al wait.

4) (0.5 puntos) PMA y PMS: Explicar el funcionamiento general de PMA y PMS. Indicar y explicar las sentencias que posee para realizar las comunicaciones básicas.

PMA: los canales son cola de mensajes enviados y aún no recibidos. Declaración de canales -> chan ch (id : tipo).

Operación send: un proceso agrega un mensaje al final de la cola ("ilimitada") de un canal ejecutando un send, que no bloquea al emisor -> send ch(var).

Operación Receive: un proceso recibe un mensaje desde un canal con receive, que demora ("bloquea") al receptor hasta que en el canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales receive ch(var);. Las variables del receive deben tener los mismos tipos que la declaración del canal. Receive es una primitiva bloqueante, ya que produce un delay -> debe recibir un mensaje para continuar con la ejecución.

PMS: los canales son de tipo link o punto a punto (1 emisor y 1 receptor).

La diferencia entre PMA y PMS es la primitiva de trasmisión Send, ya que en PMS es bloqueante:

- El trasmisor queda esperando que el mensaje sea recibido por el receptor.

- La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje -> menos memoria.
- Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (los emisores se bloquean).

Las posibilidades de deadlock son mayores.

Posee las primitivas ! para enviar mensajes (envío bloqueante y debe ser recibido antes de poder seguir). Para recibir se usa ?, que también es bloqueante.

5) (0.5 puntos) RPC y Rendezvous: Explicar las características comunes y las diferencias entre Rendezvous y RPC.

RPC: los programas se descomponen en módulos (con procesos y procedures), que pueden residir en espacios de direcciones distintos. Los procesos de 1 módulo pueden compartir variables y llamar a procedures de ese módulo. Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.

Básicamente es un protocolo que permite que un programa ejecute un procedimiento en otro espacio de memoria, como si estuviera llamando a una función local, pero en realidad, se está comunicando con un proceso remoto.

Son + adecuados para problemas de tipo manager and workers, ya que potencia la concurrencia.

Rendezvous: como con RPC, un proceso cliente invoca una operación por medio de un call, pero esta operación es servida por un proceso existente en lugar de por uno nuevo. Un proceso servidor usa una sentencia de entrada para esperar por un call y actuar. Las operaciones se atienden 1 x vez, más que concurrentemente.

Básicamente es un mecanismo de sincronización en el que los procesos esperan hasta que ambas partes estén listas para la comunicación.

Son + adecuados para problemas de tipo cliente servidor, ya que facilitan la interacción -> por su canal bidireccional.

Indicar que cosas de la comunicación guardada de rendezvous "conceptual" no se tienen en el rendezvous provisto por ADA.

En el conceptual, la cola de pedidos puedo ordenarla por ciertos factores, en ADA no.

En el modelo conceptual, puedo utilizar variables que forman parte de la entrada en la condición de la comunicación, en ADA no.

En el modelo conceptual, podría ser posible combinar el uso de una instrucción SELECT con múltiples instrucciones ACCEPT en un mismo bloque, mientras que en ADA, no se permite la combinación de un SELECT con + de 1 ACCEPT dentro del mismo bloque de sincronización.

Ventajas de las restricciones en ADA: determinismo y seguridad (asegura que el comportamiento del sistema sea predecible y libre de deadlocks inesperados), simplicidad en la implementación: la restricción en la combinación de SELECT y ACCEPT o el uso de variables de entrada facilita la implementación y el mantenimiento de sistemas, ya que el modelo es + sencillo y - propenso a errores difíciles de detectar.

Desventajas: < flexibilidad.

6) (1 punto) Librería Pthreads: explicar cómo se maneja la sincronización (ambos tipos de sincronización) en la librería Pthreads, y como se relacionan entre ellas (¿por qué para realizar una de las sincronizaciones necesita de la otra?).

Exclusión mutua: se maneja mediante variables mutex. 2 operaciones: locked y unlocked. Solo 1 hilo puede bloquear una variable, que sólo puede ser desbloqueada por quien la bloquea, es por ello que todos los mutex deben iniciarse desbloqueados. Básicamente para entrar a una SC, un Thread debe lograr tener control del mutex. Operaciones principales:

- pthread_mutex_lock(&mutex) (bloqueo)
- pthread_mutex_unlock(&mutex) (desbloqueo).

Por condición: se realiza mediante variables de condición, que permiten que un hilo se bloquee hasta que se cumpla un estado específico del programa. Similar a monitores.

Operaciones principales:

- pthread_cond_wait(pthread_cond_t cond, pthread_mutex_t *mutex): el hilo espera en la variable de condición y libera el mutex asociado.
- pthread_cond_signal(pthread_cond_t cond): notifica a un hilo en espera que la condición ha cambiado.

Una variable de condición puede asociarse a 1 o varios predicados. Si es TRUE, da 1 señal (signal) para los threads que están esperando por el cambio de estado de la condición. Si el predicado es FALSE, el thread espera en la variable condición utilizando la función pthread_cond_wait. Siempre debe bloquearse el mutex antes de testear el predicado.

La variable de condición debe tener siempre un mutex asociada a ella. Esto se debe a que en Pthreads no existen monitores como mecanismo nativo; los monitores son emulados combinando mutex y variables de condición -> en monitores estaba implícita la EM.

7) (1 punto) Librerías para Pasaje de Mensajes: Explicar los 4 diferentes tipos de SEND en que se clasifican las comunicaciones punto a punto en las librerías de Pasaje de Mensajes en general (no se refiere particularmente a MPI). Indicar cuales se corresponden con PMA y PMS de la pr 4.

SEND / RECEIVE BLOQUEANTE: no devolver el control del Send hasta que el dato a trasmitir esté seguro. Ociosidad del proceso. Hay 2 posib:

- Bloqueante con buffering: como PMA. Reduce el tiempo ocioso pero usa + memoria. El Send copia el mensaje a un buffer (que seguramente va a estar en la memoria del proceso receptor) y luego el proceso sigue la ejecución.
- Bloqueante sin buffering: PMS, el emisor y el receptor deben estar sincronizados. Genera ociosidad del emisor mientras espera que el receptor reciba el mensaje.

SEND / RECEIVE NO BLOQUEANTE: para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente. No se espera a que el dato este seguro -> requiere un posterior chequeo para asegurarse la finalización de la comunicación. 2 posib:

inseguro mientras no se copie en buffer (menos tiempo que sin buffering)

- No bloqueante con buffering: inseguro mientras no se copie en buffer (menos tiempo que sin buffering).
- No bloqueante sin buffering: hasta que el dato no haya sido recibido, la comunicación no se hace, es decir, el dato está inseguro desde el send hasta que se hace el receive.

En la práctica 4 de la materia usamos send bloqueante sin buffering (PMS) y send bloqueante con buffering (PMA).

8) (2 puntos) Paradigmas de interacción: Suponga que una imagen se encuentra representada por una matriz A(nxn), y que el valor de cada pixel es un número entero que es mantenido por un proceso distinto (es decir, el valor del pixel i, j está en el proceso P(i, j)). Cada proceso puede comunicarse a lo sumo con sus 8 vecinos que lo rodean (o la cantidad que tenga sí está en los bordes). Escriba un algoritmo heartbeat que calcule el máximo y el mínimo valor de los pixels de la imagen. Al terminar el programa, cada proceso debe conocer ambos valores.

```
chan valores[1:n,1:n](int);

Process P[i=1..n;j=1..n]

int nuevo, k, x, y, min, max, nuevoMin, nuevoMax;
int vecinos (array de tuplas (int, int)) = conocerVecinos();
int v = conocerValor();
int rondas = n-1;

for(k=1; k <= rondas; k++)
    foreach((x,y) in vecinos)
        send valores[x,y](min, max);
    foreach(vecinos)
        receive valores[i,j](nuevoMin, nuevoMax);
        if (nuevoMin < min)
            min = nuevoMin;
        if (nuevoMax > max)
            max = nuevoMax;
        end;
    end;
end;
```

CHEQUEAR

9) (2 puntos) Métricas en Sistemas Paralelos: sea la sig solución al problema del producto de matrices de nxn con P procesos en paralelo con variables compartidas. Suponga n=640 cada procesador capaz de ejecutar un proceso.

```
process worker [w = 1 to P] {
  int primera = (w-1)*(n/P) + 1;
  int ultima = primera + (n/P) - 1;

  for (i = primera to ultima) {
    for (j = 1 to n) {
       c[i][j] = 0;
       for (Z = 1 to n) {
           c[i][j] = c[i][j] +
              (a[i][Z] * b[Z][j]);
       }
    }
  }
}
```

a. Calcular cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en que P=1) y cuántas se realizan en cada procesador en la solución paralela con P=8.

b. Dados que los procesadores de P1 y P2 son idénticos, con tiempos de asignación 2, de suma 4 y de producto 8; los procesadores P3, P4, P5 y P6 son la mitad de potentes (tiempos 4, 8 y 16 para asignaciones, sumas y productos respectivamente); y los procesadores P7 y P8 son el doble de potentes que P1 (tiempos 1, 2 y 4 para asignaciones, sumas y productos respectivamente). Calcular cuánto tarda el programa paralelo y el secuencial.

c. Realizar paso a paso el cálculo del valor del speedup y la eficiencia.

d. Modificar el código para lograr una mejor eficiencia. Calcular la nueva eficiencia y comparar con la calculada en (c).

NOTA: para hacer los cálculos sólo tenga en cuenta las operaciones realizadas en la instrucción dentro del for Z.

CHEQUEAR

a- <u>En caso de que sea P = 1</u>:

Asignaciones: n^3 - Sumas: n^3 - Multiplicaciones: n^3

Total: $3n^3$

<u>En caso de que sea P = 8</u>:

Asignaciones: $n^3/8$ - Sumas: $n^3/8$ - Multiplicaciones: $n^3/8$

Total: $3n^3/8$

b- <u>Solución Secuencial</u>: Se usa P7 ->

Asignaciones: $n^3 * 1$

Sumas: $n^3 * 2$

Multiplicaciones: $n^3 * 4$

Total: $n^3 + 2n^3 + 4n^3 = 7n^3ut = 7 * 640^3 = 1.835.008.000ut$

<u>Solución Paralela</u>:

Tiempo P1..2: $2n^3/8 + 4n^3/8 + 8n^3/8 = 14/8n^3$

Tiempo P3..6: $4n^3/8 + 8n^3/8 + 16n^3/8 = 28/8n^3$

Tiempo P7..8: $n^3/8 + 2n^3/8 + 4n^3/8 = 7/8n^3$

Tiempo paralelo: max (TiempoP 1..2, TiempoP 3..6, TiempoP 7..8) = $28/8n^3 = 7/2n^3$

c- <u>Speedup</u>: $T_{seq}/T_{par} = 7n^3 / (7/2n^3) = 7 * 7/2 = 2$

<u>$S_{óptimo}$</u> = $2 * [(7/8n^3) / (14/8n^3)] + 4 * [(7/8n^3) / (28/8n^3)] + 2 * [(7/8n^3) / (7/8n^3)] = 1 + 1 + 2 = 4$

<u>Eficiencia</u>: Speedup / $S_{óptimo}$ = 2 / 4 = 50%

d- <u>Modificación del código</u>:

```
-- Factor es un array que conoce el factor de
-- carga calculado arriba para cada procesador
process worker[w=1 to P]{
    int primera = (w-1)*n*factor[w]+1
    int ultima = primera+n*factor[w]-1

    for [i = primera to última]{
        for [j=1 to n]{
            c[i,j]=0
            for [Z=1 to n]{
                c[i,j]=c[i,j]+(a[i,Z]*b[Z,j])
            }
        }
    }
}
```

$4 * x + 2 * 2x + 2 * 4x = 1$

$4x + 4x + 8x = 1$

$x = 1/16$ siendo x la unidad mínima de trabajo

Carga P1..2: 2*1/16 = 1/8

Carga P3..6: 1/16

Carga P7..8: 4*1/16 = ¼

Tiempo P1..2: $2n^3 / 8 + 4n^3 / 8 + 8n^3 / 8 = 14 / 8n^3 = 7/4n^3$

Tiempo P3..6: $4n^3 / 16 + 8n^3 / 16 + 16n^3 / 16 = 28/16n^3 = 7/4n^3$

Tiempo P7..8: $n^3 / 4 + 2n^3 / 4 + 4n^3 / 4 = 7/4n^3$

Tiempo paralelo: $7/4n^3$.

Speedup: $T_{seq} / T_{par} = 7n^3 / (7/4n^3) = 7 * 7/4 = 4$

Eficiencia: Speedup / $S_{óptimo}$ = 4/4 = 100%